CHAPTER 1

What is Performance

Performance refers to the speed at which an application functions. It is a multifaceted aspect of quality. When we're talking about web applications, the time it takes your application to be presented to your users is what we will call *web performance*. The speed at which your application responds to your users' interactions is what we'll call *runtime performance*. These are the two facets of performance that we will be looking at.

Performance in the context of web (and especially mobile web) development is a relatively new subject, but it is absolutely overdue for the attention it has been getting.

In this book we will explore how to quantify and optimize JavaScript performance, in the context of both web performance and runtime performance. This is vitally important because JavaScript is potentially the largest area for improvement when trying to address the total performance of your site. Steve Souders, architect of both YSlow and PageSpeed, and pioneer in the world of web performance, has demonstrated this point in an experiment where he showed an average performance improvement of 31% when removing JavaScript from a sample of web sites.[1] We can completely remove any JavaScript from our site as Steve did in his experiment, or we can refine how we write JavaScript and learn to measure the efficiencies in what we write.

It's not realistic to remove JavaScript from our front-end, so let's look at making our JavaScript more efficient. Arguably even more important, let's look at how we can create automated tools to track these efficiencies and visualize them for reporting and analysis.

Web Performance

Sitting with your laptop or holding your device, you open a web browser, type in a URL and hit Enter, and wait for the page to be delivered to and rendered by your browser. The span of time that you are waiting for the page to be usable depends on web performance. For our purposes we will define web performance as an overall indicator of the time it takes for a page to be delivered and made available to your end user.

There are many things that influence web performance, network latency being the first. How fast is your network? How many round trips and server responses are needed to serve up your content?

To better understand network latency, let's first look at the steps in an HTTP transaction (Figure 1.1).

When it requests a URL, whether the URL for a web page or a URL for each asset on a web page, the browser spins up a thread to handle the request and initiates a DNS lookup at the remote DNS server. This allows the browser to get the IP address for the URL entered.

1 `http://www.stevesouders.com/blog/2012/01/13/javascript-performance/`

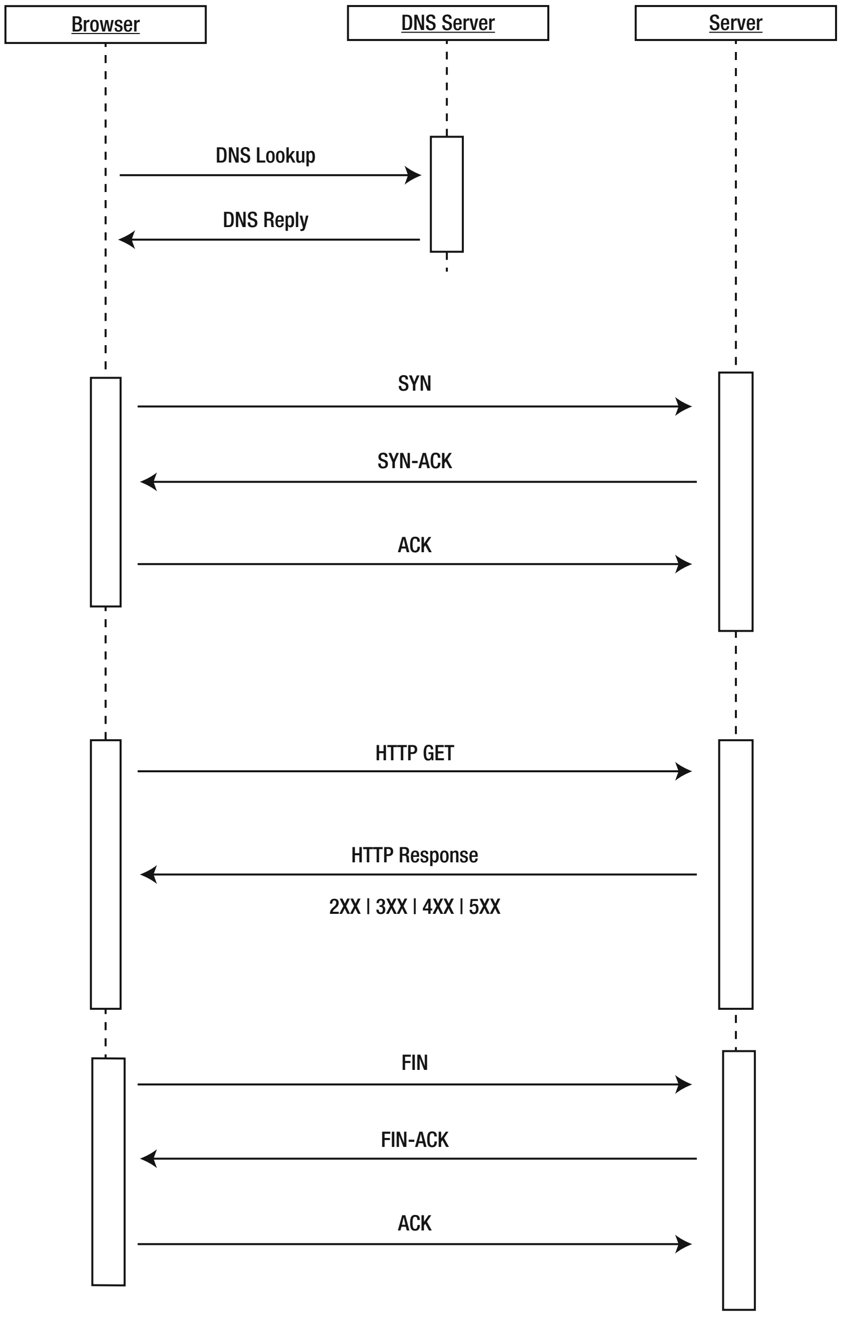


Figure 1-1. *Sequence diagram of network transactions in a request for a web page and repeated for each remote -object included in a web page*

■ **Note** *Threads* are sequential units of controlled execution for applications. Whenever an application performs any operation, it uses a thread. Some applications are multithreaded, which means that they can do multiple things at once. Generally browsers use at least one thread per tab. That means that the steps that the thread executes—the steps that we outline as part of the connection, download and rendering process—are handled sequentially.

Next the browser negotiates a TCP three-way handshake with the remote web server to set up a TCP/IP connection. This handshake consists of a Synchronize, Synchronize-Acknowledge, and Acknowledge message to be passed between the browser and the remote server. This handshake allows the client to attempt communication, the server to acknowledge and accept the attempt, and the client to acknowledge that the attempt has been accepted.

This handshake is much like the military voice procedure for two way radio communication. Picture two parties on either end of a two way radio—how do they know when the other party has finished their message, how do they know not to talk over each other, and how do they know that the one side understood the message from the other? These have been standardized in voice procedure, where certain key phrases have nuanced meaning; for example, Over means that one party has finished speaking and is waiting for a response, and Roger indicates that the message has been understood.

The TCP handshake, like all communication protocols, is just a standardized way to define communication between multiple parties.

THE TCP/IP MODEL

TCP stands for Transmission Control Protocol. It is the protocol that is used in the TCP/IP model that defines how communications between a client and a server are handled, specifically breaking the data into segments, and handling the handshake that we described earlier (Figure 1.1).

The TCP/IP model is a four-layer model that represents the relationship between the different protocols that define how data is shared across the Internet. The specification for the TCP/IP model is maintained by the Internet Engineering Task Force, in two RFC (Request For Comment) documents, found here: `http://tools.ietf.org/html/rfc1122` and `http://tools.ietf.org/html/rfc1123`.

The four layers in the TCP/IP model are, in order from furthest to closest to the end user, the Network Access layer, the Internet layer, the Transport layer, and the Application layer.

The Network Access layer controls the communication between the hardware in the network.

The Internet layer handles network addressing and routing, getting IP and MAC addresses.

The Transport layer is where our TCP (or UDP) communication takes place.

The Application layer handles the top-level communication that the client and servers use, like HTTP and SMTP for email clients.

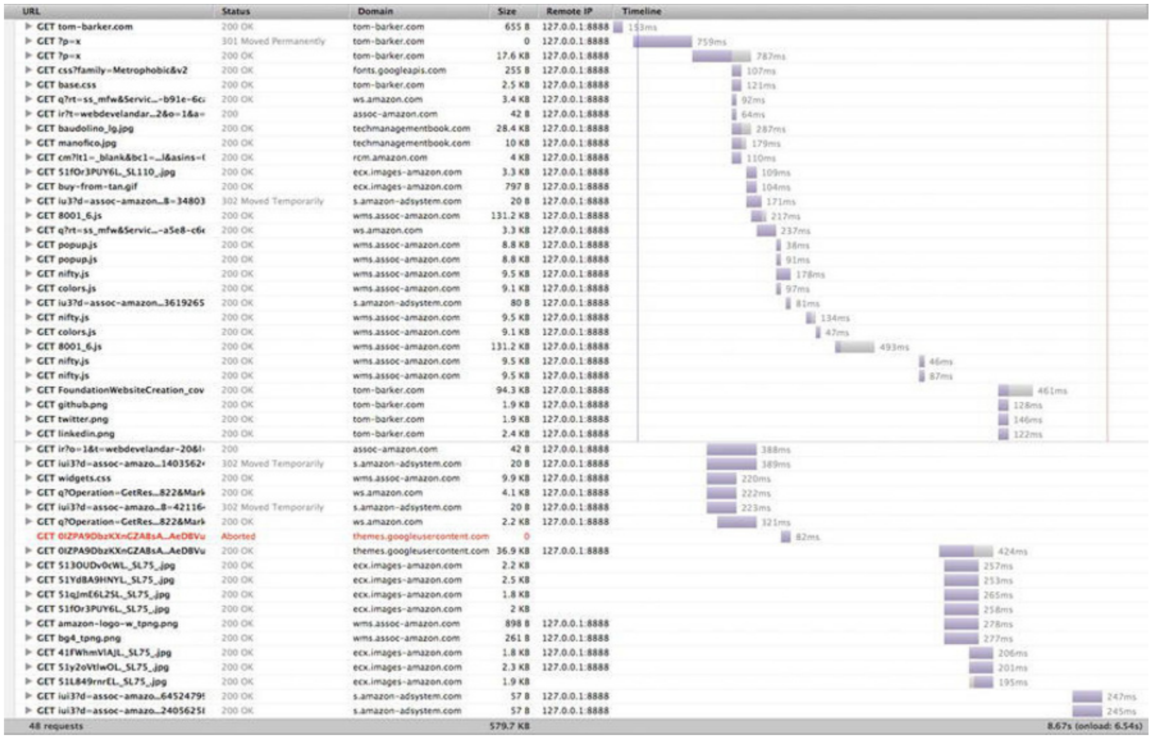
If we compare the TCP/IP model to our sequence diagram, we see how the browser must traverse up and down the model to serve up our page, as shown here.

Once the TCP/IP connection has been established, the browser sends an HTTP GET request over the connection to the remote server. The remote server finds the resource and returns it in an HTTP Response, the status of which is 200 to indicate a good response. If the server cannot find the resource or generates an error when trying to interpret it, or if the request is redirected, the status of the HTTP Response will reflect these as well. The full list of status codes can be found at `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html` but the most common ones are these:

- 200 indicates a successful response from the server.
- 404 means that the server could not find the resource requested.
- 500 means that there was an error when trying to fulfill the request.

It is here that the web server serves up the asset and the client begins downloading it. It is here that the total payload of your page—which includes file sizes of all images, CSS, and JavaScript—comes into play.

The total size of the page is important, not just because of the time it takes to download, but because the maximum size of an IP packet is 65535 octets for IPv4 and IPv6. If you take your total page size converted to bytes and divide it by the maximum packet size, you will get the number of server responses needed to serve up your total payload.



URL	Status	Domain	Size	Remote IP	Timeline
▶ GET tom-barker.com	200 OK	tom-barker.com	655 B	127.0.0.1:8888	153ms
▶ GET ?p=x	301 Moved Permanently	tom-barker.com	0	127.0.0.1:8888	759ms
▶ GET ?p=x	200 OK	tom-barker.com	17.6 KB	127.0.0.1:8888	787ms
▶ GET css?family=Metrophobic&v2	200 OK	fonts.googleapis.com	255 B	127.0.0.1:8888	107ms
▶ GET base.css	200 OK	tom-barker.com	2.5 KB	127.0.0.1:8888	121ms
▶ GET q?rt=ss_mfw&Servic...-b91e-6c	200 OK	ws.amazon.com	3.4 KB	127.0.0.1:8888	92ms
▶ GET ir?t=webdevelandar...2&o=1&a=	200	assoc-amazon.com	42 B	127.0.0.1:8888	64ms
▶ GET baudolino_lg.jpg	200 OK	techmanagementbook.com	28.4 KB	127.0.0.1:8888	287ms
▶ GET manofico.jpg	200 OK	techmanagementbook.com	10 KB	127.0.0.1:8888	179ms
▶ GET cm?lt1=_blank&bc1=...l&asins=(	200 OK	rcm.amazon.com	4 KB	127.0.0.1:8888	110ms
▶ GET 51fOr3PUY6L._SL110_.jpg	200 OK	ecx.images-amazon.com	3.3 KB	127.0.0.1:8888	109ms
▶ GET buy-from-tan.gif	200 OK	ecx.images-amazon.com	797 B	127.0.0.1:8888	104ms
▶ GET iu3?d=assoc-amazon...8=34803	302 Moved Temporarily	s.amazon-adsystem.com	20 B	127.0.0.1:8888	171ms
▶ GET 8001_6.js	200 OK	wms.assoc-amazon.com	131.2 KB	127.0.0.1:8888	217ms
▶ GET q?rt=ss_mfw&Servic...-a5e8-c6	200 OK	ws.amazon.com	3.3 KB	127.0.0.1:8888	237ms
▶ GET popup.js	200 OK	wms.assoc-amazon.com	8.8 KB	127.0.0.1:8888	38ms
▶ GET popup.js	200 OK	wms.assoc-amazon.com	8.8 KB	127.0.0.1:8888	91ms
▶ GET nifty.js	200 OK	wms.assoc-amazon.com	9.5 KB	127.0.0.1:8888	178ms
▶ GET colors.js	200 OK	wms.assoc-amazon.com	9.1 KB	127.0.0.1:8888	97ms
▶ GET iu3?d=assoc-amazon...3619265	200 OK	s.amazon-adsystem.com	80 B	127.0.0.1:8888	81ms
▶ GET nifty.js	200 OK	wms.assoc-amazon.com	9.5 KB	127.0.0.1:8888	134ms
▶ GET colors.js	200 OK	wms.assoc-amazon.com	9.1 KB	127.0.0.1:8888	47ms
▶ GET 8001_6.js	200 OK	wms.assoc-amazon.com	131.2 KB	127.0.0.1:8888	493ms
▶ GET nifty.js	200 OK	wms.assoc-amazon.com	9.5 KB	127.0.0.1:8888	46ms
▶ GET nifty.js	200 OK	wms.assoc-amazon.com	9.5 KB	127.0.0.1:8888	87ms
▶ GET FoundationWebsiteCreation_cov	200 OK	tom-barker.com	94.3 KB	127.0.0.1:8888	461ms
▶ GET github.png	200 OK	tom-barker.com	1.9 KB	127.0.0.1:8888	128ms
▶ GET twitter.png	200 OK	tom-barker.com	1.9 KB	127.0.0.1:8888	146ms
▶ GET linkedin.png	200 OK	tom-barker.com	2.4 KB	127.0.0.1:8888	122ms
▶ GET ir?o=1&t=webdevelandar-20&l=	200	assoc-amazon.com	42 B	127.0.0.1:8888	388ms
▶ GET iui3?d=assoc-amazo...1403562	302 Moved Temporarily	s.amazon-adsystem.com	20 B	127.0.0.1:8888	389ms
▶ GET widgets.css	200 OK	wms.assoc-amazon.com	9.9 KB	127.0.0.1:8888	220ms
▶ GET q?Operation=GetRes...822&Mark	200 OK	ws.amazon.com	4.1 KB	127.0.0.1:8888	222ms
▶ GET iui3?d=assoc-amazo...8=42116	302 Moved Temporarily	s.amazon-adsystem.com	20 B	127.0.0.1:8888	223ms
▶ GET q?Operation=GetRes...822&Mark	200 OK	ws.amazon.com	2.2 KB	127.0.0.1:8888	321ms
GET 0IZPA9DbzKXnGZA8sA...AeD8Vu	Aborted	themes.googleusercontent.com	0		82ms
▶ GET 0IZPA9DbzKXnGZA8sA...AeD8Vu	200 OK	themes.googleusercontent.com	36.9 KB	127.0.0.1:8888	424ms
▶ GET 513OUDv0cWL._SL75_.jpg	200 OK	ecx.images-amazon.com	2.2 KB		257ms
▶ GET 51YdBA9HNYL._SL75_.jpg	200 OK	ecx.images-amazon.com	2.5 KB		253ms
▶ GET 51qJmE6L2SL._SL75_.jpg	200 OK	ecx.images-amazon.com	1.8 KB		265ms
▶ GET 51fOr3PUY6L._SL75_.jpg	200 OK	ecx.images-amazon.com	2 KB		258ms
▶ GET amazon-logo-w_tpng.png	200 OK	wms.assoc-amazon.com	898 B	127.0.0.1:8888	278ms
▶ GET bg4_tpng.png	200 OK	wms.assoc-amazon.com	261 B	127.0.0.1:8888	277ms
▶ GET 41FWhmVIAJL._SL75_.jpg	200 OK	ecx.images-amazon.com	1.8 KB	127.0.0.1:8888	206ms
▶ GET 51y2oVtlwOL._SL75_.jpg	200 OK	ecx.images-amazon.com	2.3 KB	127.0.0.1:8888	201ms
▶ GET 51L849rnrEL._SL75_.jpg	200 OK	ecx.images-amazon.com	1.9 KB		195ms
▶ GET iui3?d=assoc-amazo...6452479	200 OK	s.amazon-adsystem.com	57 B	127.0.0.1:8888	247ms
▶ GET iui3?d=assoc-amazo...2405625	200 OK	s.amazon-adsystem.com	57 B	127.0.0.1:8888	245ms
48 requests			579.7 KB		8.67s (onload: 6.54s)

Figure 1-2. *Browser architecture*

Another contributor to network latency is the number of HTTP requests that your page needs to make to load all of the objects on the page. Every asset that is included on the page—each image and external JavaScript and CSS file—requires a round trip to the server. Each spins up a new thread and a new instance of the flow shown in Figure 1-1, which again includes a cost for DNS lookup, TCP connection, and HTTP request and response, plus the cost in time transmitting the sheer file size of each asset.

See Figure 1-2 for an idea of how this simple concept can exponentially grow and cause performance hits in scale.

Waterfall charts are a tool to demonstrate the time it takes to request a page and all of the assets included in the page. They show the HTTP transaction for each asset needed to construct a page, including the size of each asset, how long each one took to download, and the sequence in which they were downloaded. At a high level, each bar in the waterfall chart is a resource that we are downloading. The length of a bar corresponds to how long an item takes to connect to and download. The chart runs on a sequential timeline, so that the top bar is the first item that gets downloaded and the last bar is the final item, and the far left of the timeline is when the connections begin and the far right is when they end. We will talk much more about waterfall charts in Chapter 2, when we discuss tools for measuring and impacting performance.

Parsing and Rendering

Another influencer of web performance, outside of network concerns, is browser parsing and rendering. Browser parsing and rendering is influenced by a number of things. To better understand this concept let's first look at an overview of the browser's architecture as it pertains to parsing and rendering web pages (Figure 1-3).

Most modern browsers have the following architecture: code to handle the UI, including the location bar and the history buttons, a Rendering Engine for parsing and drawing all of the objects in the page, a JavaScript Engine for interpreting the JavaScript, and a network layer to handle the HTTP requests.

Since the browser reads content from the top down, where you place your assets impacts the perceived speed of your site. For example, if you put your JavaScript tags before HTML content, the browser will launch the JavaScript interpreter and parse the JavaScript before it finishes rendering the remainder of the HTML content, which can delay making the page usable for the end user.

Browsers are your bread and butter as a web developer, and so you should be more than familiar with each of the rendering engines and JavaScript engines. It is more than worth your time to download the

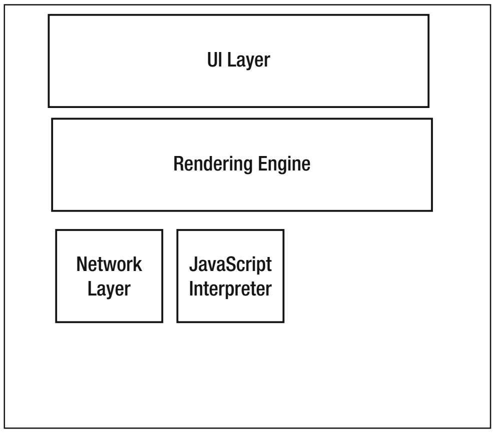


Figure 1-3. *Time series of my lift log*

ones that are open-source (see the next section for URLs where available) and read through some of the source code. If you are really adventurous you can put your own instrumentation or logging into the source code and automate your own performance tests running in your forked engine.

Rendering Engines

Let's take a look at some of the more widely used rendering engines out in the wild. It's important to think of the rendering engine as more than the browser. By modularizing the architecture of the browsers, the browser makers have been able to federate the components. More tools than just browsers render HTML, including email clients and web components in other applications. By having a distributable rendering engine, browser makers can reuse their own engines or license them for use by other companies. This also usually allows developers to know what to expect from a software package just by knowing which rendering engine it is using.

Firefox and all of its derivatives and cousins (like Thunderbird, Mozilla's email client) use Gecko, available at `https://developer.mozilla.org/en/Gecko`. Gecko was first developed at Netscape, before the Mozilla Project spun out as its own entity, as the successor to the original Netscape rendering engine, back in 1997.

Webkit is what Chrome and Safari use, and is your target for most mobile web development since it is used as the layout or rendering engine for Android devices as well as mobile Safari for iOS devices and the Silk browser on Kindle Fires. Webkit is available at `http://www.webkit.org/`. WebKit was started in 2001 at Apple as a fork of a previous rending engine, KHTML from KDE. WebKit was open sourced publicly in 2005.

Opera on desktop, mobile, and even all the Nintendo consoles (NDS, Wii) use Presto, which was introduced in 2003 with Opera 7. More information about Presto can be found at `http://dev.opera.com/articles/view/presto-2-1-web-standards-supported-by/`.

And finally, Internet Explorer, along with other Microsoft products like Outlook, uses MSHTML, codenamed Trident. Microsoft first introduced Trident with Internet Explorer 4 in 1997 and has been iterating on the engine since. Documentation for Trident can be found here: `http://msdn.microsoft.com/en-us/library/bb508515`.

JavaScript Engines

Next let's take a look at the JavaScript engines used by the most popular browsers. Modularizing the JavaScript interpreter makes the same kind of sense as modularizing the rendering engine, or modularizing any code for that matter. The interpreter can be shared with other properties, or embedded in other tools. The open source interpreters can even be used in your own projects, perhaps to build your own static code analysis tools, or even just to build in JavaScript support to allow your users to script certain functionality in your applications.

SpiderMonkey is the JavaScript engine made by Mozilla that is used in Firefox. Brendan Eich, creator of JavaScript, created SpiderMonkey in 1996 and it has been the JavaScript interpreter for Netscape and then Firefox ever since. The documentation for SpiderMonkey is available here: `https://developer.mozilla.org/en/SpiderMonkey`. Mozilla has provided documentation showing how to embed SpiderMonkey into our own applications here: `https://developer.mozilla.org/en/How_to_embed_the_JavaScript_engine`.

Opera uses Carakan, which was introduced in 2010. More information about Carakan can be found here: `http://my.opera.com/dragonfly/blog/index.dml/tag/Carakan`.

Google's open source JavaScript Engine used by Chrome is available here: `http://code.google.com/p/v8/`. Documentation for it is available here: `https://developers.google.com/v8/intro`.

Safari uses JavaScriptCore, sometimes called Nitro. More information about JavaScriptCore can be found here: `http://www.webkit.org/projects/javascript/index.html`.

And finally, Internet Explorer uses Chakra as their JScript engine. Remember that, as Douglas Crockford details at `http://www.yuiblog.com/blog/2007/01/24/video-crockford-tjpl/`, JScript started life as Microsoft's own reverse-engineered version of JavaScript. Microsoft has since gone on to give JScript its own voice in the overall ecosystem. It is a legitimate implementation of the ECMAScript spec, and Chakra even supports some aspects of the spec that most other JavaScript engines don't, specifically conditional compilation (see the accompanying discussion of conditional compilation).

All of these are nuances to consider when talking about and optimizing the overall web performance of your site.

The JavaScript team at Mozilla also maintains a site, `http://arewefastyet.com/`, that compares benchmarking times for V8 and SpiderMonkey, comparing the results of both engines running the benchmarking test suites of each engine.

CONDITIONAL COMPILATION

Conditional compilation is a feature of some languages that traditionally allows the language compiler to produce different executable code based on conditions specified at compile time. This is somewhat of a misnomer for JavaScript because, of course, JavaScript is interpreted, not compiled (it doesn't run at the kernel level but in the browser), but the idea translates.

Conditional compilation allows for writing JavaScript that will only be interpreted if specific conditions are met. By default conditional compilation is turned off for JScript; we need to provide an interpreter-level flag to turn it on: `@cc_on`. If we are going to write conditionally compiled JavaScript, we should wrap it in comments so that our code doesn't break in other JavaScript interpreters that don't support conditional compilation.

An example of JScript conditional compilation is

```
<script>

var useAX = false; //use ActiveX controls default to false

/*@cc_on

@if (@_win32)

     useAX = true;

@end

*/

</script>
```

Runtime Performance

Runtime is the duration of time that your application is executing, or running. Runtime performance speaks to how quickly your application responds to user input while it is running—for example, while saving preferences, or when accessing elements in the DOM.

Runtime performance is influenced by any number of things—from the efficiency of the algorithms employed for specific functionality, to optimizations or shortcomings of the interpreter or browser rendering engine, to effective memory management and CPU usage, to design choices between synchronous or asynchronous operations.

While runtime performance is thus a subjective perception of the overall peppiness of your application, you can build in instrumentation that will allow you to track the shape and trend of your users' overall experiences and analyze the outliers. You can also conduct multivariate testing experiments to see what approach yields the greatest performance gain at scale and with the browsers in use with your specific user base.

We will explore these ideas in Chapter 4.

Why does performance matter?

The first reason should be obvious—faster web sites mean a better overall user experience for your end user. A better experience in theory should equate to happier users.

A faster experience also means that users can access your features faster, hopefully before they abandon the session. Session or site abandonment happens for any number of reasons: pages taking too long to load, users losing interest, browsers crashing, or any other of a near-infinite number of reasons.

Figuring out your own site abandonment rate is easy. Just take the total number of users who do whatever action you want of them—purchase an item, register a new account, upsell to a service, view pages in other sections, click a given button on the homepage, whatever the high-level objective is that you have for your site. You take that number and divide it by the total number of visits. Subtract that from one and multiply that by 100 to give you the percentage of traffic that abandoned your site before fulfilling your objective:

```
[abandonment rate] = (1 - ([number of fulfilled objectives] \ [total number of visits])) * 100
```

As an example, say we have a web form, maybe a customer registration page. The whole point of that page is to get users to create accounts—once they have an account we can start tailoring things to their own personal preferences, we can target ads to their purchasing habits, and we can make recommendations to them based on past purchases and viewing history. Whatever the purpose, we want them signed in and that's how we'll measure the success of this page. Once a user hits Submit on the form, we go to a PHP script that updates a database, creates a new entry in our User table, and then directs to our homepage.

So we look at the page view metrics for this page and see that we have 100,000 unique page views; in our algorithm this is the total number of visits. If we look at the number of users created in our database, we see that we have 30,000 users. At this point we could apply the algorithm to get our abandonment rate of 70%:

```
(1 - (30,000 \ 100,000)) * 100 = 70
```

Improving performance can bring significant benefits to your bottom line by reducing your abandonment rate. There have been a number of prominent case studies where companies have demonstrated the tangible harm (seen in increased abandonment rates) caused by poor web performance.

Keynote has made available an article by Alberto Savoia, detailing the impact of performance on abandonment rates at `http://www.keynote.com/downloads/articles/tradesecrets.pdf`. In their whitepaper "Why Web Performance Matters," available at `http://www.gomez.com/pdfs/wp_why_web_performance_matters.pdf`, Gomez details how abandonment rates can increase from 8% up to 38% just by introducing latency in page web performance.

You can run your own experiments using the calculation just shown to quantify and extrapolate the return on investment for optimizing site performance.

Instrumentation and Visualization

A big part of this book is about putting tooling in your code and using data visualizations to demonstrate the results. In truth, that is kind of the point of this book. There is no one silver-bullet solution when it comes to performance. The results that one person sees may not be the same results that another gets, because they may have a completely different user base, using a completely different browser.

Maybe your users are locked into using Internet Explorer because of corporate policy, or maybe your audience is made up of early adopters and you have a high population of people using beta releases of browsers, which may have different optimizations in their interpreter or rendering engine, or may even have bugs in their interpreter or rendering engine.

Whatever the case, your results will vary. And they will vary at scale, because of connection speed at different times of the day (users at work versus users at home), because of their method of connecting (cable versus dial up), or any other reason.

But by measuring your own results and visualizing them to see the overall shape of what your data looks like, you'll be able to fine-tune your own site based on your own real data and trends.

Data visualization as a discipline has blossomed lately. No longer is it relegated solely to the world of mathematics, theory, or cartography. I remember when I first got an inkling of what I could do with data visualization. I was at a conference; it was Velocity in Santa Clara surrounded by my peers. I watched John Rauser give a talk about how he and his team at Amazon debug production issues by analyzing production logs. In his session he talked about sometimes needing to pull out granular data at the individual user level, lay it out in hard copy, and just squint at the data to see the overall shape of it. The shape is what was telling.

That really resonated with me, and since then I've explored that in almost every aspect of my life.

At work I use data visualizations as management tools for running my organization. Some of the charts that we will be creating in this book are derived from charts that I regularly run for my own team.

In my leisure time I trend my power lifting lift log to see my increases, my resets, and when I plateau (see Figure 1-4). I can see how other things going on in my life affect my lift increases, by cross-referencing dates in the time series. Data analysis is actually a key concept in power lifting, enabling you to manage your increases in weight by measuring your recover time. The sign that you have advanced to a higher level of experience is the time it takes to recover from heavy lifts and the increase in the amount that you are lifting. Beginners advance very quickly because they are lifting far from their potential weight ceiling, but intermediate and advanced lifters push their muscles so hard and work so close to their potential ceiling that it takes them much longer to recover and increase their lift weights.[2]

At home I also track the humidity level in each room of my house, and I play with the dials. I see what effect running the heat has on the humidity, or caulking the spaces between the floorboards and the walls, or even just having the doors open instead of closed for each room in the house. In such a way I can aspire to naturally have the lowest possible humidity level in my house without running my dehumidifier.

Visualizing my data allows me to see a larger scope of a situation and to clearly see any spikes, outliers, or trends that might not be obvious in the raw data.

2 See Mark Rippetoe's *Starting Strength* (Aasgard Press)

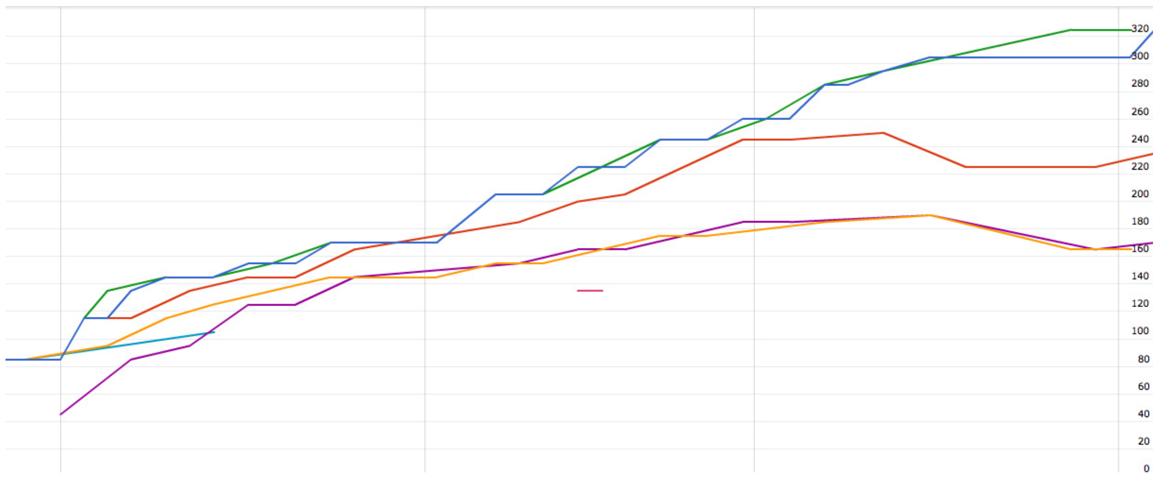


***Figure 1-4** Time series of my lift log*

The Goal of This Book

There is no shortage of information available online and in other books about current best practices for performance—but performance is a moving target. Because each browser uses a different JavaScript interpreter and rendering engine, your results will differ between browsers and browser versions. Best practices are changing or becoming redefined continually because of changes and optimizations at the interpreter level, differences in system configuration, and network speeds. This pace of change is exacerbated by the quickened release schedule that most browsers have adopted.

But just as important as following best practices is the ability to measure your own performance, so that you can adjust as times change, and so that you can note the subtle nuances in your own code and define your own best practices by your own observations.

My goal with this book is to give you the tools to observe and track over time the performance of your web applications from multiple perspectives, so that you are always aware of all aspects of your performance. And by tools, I don't just mean the code that we will develop through the course of the book or the applications available that we will talk about and even automate. I mean the insight to care about these metrics and the mental models to build such instrumentation and visualization into everything that you do.

In many ways, analyzing and optimizing the efficiency of how things operate and perform is part of reaching the next level of excellence. Any journeyman can create something to spec, but a master crafts with excellence and proves that excellence with empirical data.

Technologies Used and Further Reading

As the title suggests, we use JavaScript extensively throughout this book. We also use PHP to automate certain tools, scrape results, and format data. If you aren't already familiar with PHP, its grammar and lexicon are fairly similar to JavaScript, so you should have no problem switching context between the two languages. Extensive coverage of PHP is outside the scope of this book. If you want more of an introduction to the language you can check out *Beginning PHP and MySQL*, by W. Jason Gilmore (Apress, 2005), or if you want a deeper dive into modern PHP, check out *Pro PHP Programming*, by Peter MacIntyre, Brian Danchilla, and Mladen Gogala (Apress, 2011).

Another language we will use quite a bit is R, which is both a language and the environment that runs the language, and it is used to run statistical calculations and chart data that you import or derive. It is a very interesting language with a very specific use.

R can be daunting at first if you aren't familiar with its syntax or even things as fundamental as its different data types. Don't worry; I will explain everything that you need to know to understand the code that we will be writing in R. If you'd like a deeper dive into R—and with most statistical information from the top companies being derived in R,[3] and data science being one of the largest growth fields in the coming years,[4] why wouldn't you want to know more about R?—then I recommend *R in Action*, by Robert Kabicoff (Manning, 2011) and *The Art of R Programming: A Tour of Statistical Design*, by Norman Matloff (No Starch Press, 2011). Both books approach R as a programming language, as opposed to a mathematical environment, which makes it easier for developers to grasp.

R is amazingly useful to learn, and the more you use it the more you'll find uses for it. And it's completely extensible, with a rich plugin architecture and a huge community that builds plugins; it's rare to find something that R can't do—at least in the realm of statistics and data visualization.

As I said earlier, there are many resources available for further reading and exploration on the subject of overall web performance optimization. I've referenced Steve Souders' works already; he is a luminary in the field of web performance. His web site is `http://www.stevesouders.com/` and he has written two books that go deep into many aspects of web performance. He also runs `http://httparchive.org/`, whose goal is to be an archive of performance metrics and statistics for the web. All manner of interesting things are to be found here, from the percentage of the web using JQuery to the overall trend of Flash usage over time. This is hugely useful for seeing overall trends as well as doing competitive analysis when developing new features or applications.

The Worldwide Web Consortium (W3C) has a working group dedicated to web performance. This group is working to create specifications and extensions to current standards to expose functionality that will give developers more control in tracking performance natively in a browser. Their charter is located here: `http://www.w3.org/2010/webperf/`. We will be discussing the progress and specifications that have come from this group in Chapter 5.

Since the point of this book is not just about performance but also about visualizing information, I recommend Nathan Yau's book *Visualize This: The FlowingData Guide to Design, Visualization, and Statistics* (Wiley, 2011) as a great primer for data visualization as a craft. Nathan also maintains `http://flowingdata.com/`.

Summary

This chapter explored some introductory concepts around performance. We defined two aspects of performance for web applications; web performance is an indication of the time it takes to serve content to our end users, and runtime performance is an indication of how responsive our applications are while our end users are using them.

We briefly explored some of the protocols that hold the web together, like the TCP/IP model, and we traced a request for content from our browser up the TCP/IP model, correlating each action along the way with where in the model it was taking place. We examined the architecture of a TCP round trip and saw the steps involved that our browsers need to take for every piece of content that we request—sometimes in the case of HTTP redirects, multiple times for each of content.

---

3 `http://www.revolutionanalytics.com/what-is-open-source-r/companies-using-r.php` and `http://www.nytimes.com/2009/01/07/technology/business-computing/07program.html`

4 `http://mashable.com/2012/01/13/career-of-the-future-data-scientist-infographic/`

We looked at modern browser architecture and saw that browsers are no longer huge black-box monoliths, but instead are modular and some even open source. We talked about the benefits of this modular architecture, noting that as the web becomes ubiquitous, rendering engines are being used for other applications to parse and render markup in email clients or embedded in custom applications, and that we can even embed pieces of browsers in our own applications.

We looked at why performance matters to our business, from customer happiness to looking at abandonment rates.

Finally we started to talk about gathering, analyzing, and visualizing our data. This last point is a recurring theme that we will see throughout this book—measuring and quantifying with empirical data, visualizing that data to show the overall shape of the data. The shape of the data is key; it can reveal trends and patterns that aren't obvious in the raw data. We can look at a visualization immediately and know generally what it is saying.

We'll look much deeper into these concepts in the coming chapters, and we begin in the next chapter by exploring tools that are available for us to track and improve performance.